PDist-RIA Crawler: A Peer-to-Peer Distributed Crawler for Rich Internet Applications

Seyed M. Mirtaheri[1], Gregor V. Bochmann[1], Guy-Vincent Jourdan[1], and Iosif Viorel Onut[2]

[1] School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Ontario, Canada
staheri@uottawa.ca, {gvj,bochmann}@eecs.uottawa.ca
[2] Security AppScan® Enterprise, IBM
770 Palladium Dr, Ottawa, Ontario, Canada
vioonut@ca.ibm.com

Abstract. Crawling Rich Internet Applications (RIAs) is important to ensure their security, accessibility and to index them for searching. To crawl a RIA, the crawler has to reach every application state and execute every application event. On a large RIA, this operation takes a long time. Previously published *GDist-RIA Crawler* proposes a distributed architecture to parallelize the task of crawling RIAs, and run the crawl over multiple computers to reduce time. In GDist-RIA Crawler, a centralized unit calculates the next task to execute, and tasks are dispatched to worker nodes for execution. This architecture is not scalable due to the centralized unit which is bound to become a bottleneck as the number of nodes increases. This paper extends GDist-RIA Crawler and proposes a fully peer-to-peer and scalable architecture to crawl RIAs, called *PDist-RIA Crawler*. PDist-RIA doesn't have the same limitations in terms scalability while matching the performance of GDist-RIA. We describe a prototype showing the scalability and performance of the proposed solution.

Keywords: Web Crawling, Rich Internet Application, Peer-to-Peer Algorithm, Crawling Strategies.

1 Introduction

Crawling a web application refers to the process of discovering and retrieving client-side application states. Traditionally, a web crawler finds the initial state of the application through its URL, referred to as *seed URL*. The crawler parses this page, finds new URLs that belong to the application, and retrieves them. This process continues recursively until all states of the application are discovered.

Unlike traditional web applications, in modern web applications (referred to as *Rich Internet Applications* or RIAs), different states of the application are not always reachable through URLs. When the user interacts with the application locally, the client-side of the application may or may not interact with the server, and different states of the application are constructed on the client. In this realm,

B. Benatallah et al. (Eds.): WISE 2014, Part II, LNCS 8787, pp. 365–380, 2014.

it is no longer sufficient to discover every application URL. To crawl a RIA, the crawler has to execute all events in all states of the application.

In effect, to crawl a RIA, the crawler emulates a user session. It loads the RIA in a virtual web browser, interacts with the application by triggering user interface events such as clicking on buttons or submitting forms [2, 7, 11, 20]. Occasionally, the crawler cannot reach a target state from its current state merely by interacting with the website. When this is the case, the crawler has to reload the seed URL. Through interaction with the application, the crawler discovers all application states.

For a large RIA, executing all events is a very time-consuming task. One way to reduce the time it takes to crawl RIAs is to parallelize the crawl and run it on multiple computers (henceforth referred to as *nodes*). Parallel crawling of RIAs was first explored by *Dist-RIA Crawler* [24]. It was proposd to run the crawl in parallel on multiple nodes with a centralized unit called *coordinator*. Dist-RIA Crawler partitions the task of crawling a RIA by assigning different events to different nodes. In this algorithm, all nodes visit all application states, however, each node is only responsible for the execution of a subset of the events in each state. Together the nodes execute all events on all states. When a node discovers a new state, it informs the coordinator, and the coordinator informs other nodes about the new state. The coordinator is also the one detecting termination.

In the context of crawling RIAs, the *crawling strategy* refers to the strategy that the crawler uses to choose the next event to execute [7]. Efficiency of crawling is effected by the crawling strategy. Two of the most efficient crawling strategies are the greedy [26] and the probabilistic [10] strategies. The greedy strategy finds the un-executed event which is the closest to the current state of the crawler. The probabilistic strategy, uses the history of event executions and chooses an event that maximized the likelihood of finding a new state. Running simple strategies such as breath-first and depth-first search does not require knowledge of all application graph transitions. To run efficient strategies, however, it is crucial to have access to all known application graph transitions.

In Dist-RIA Crawler only the application states are sent to the other nodes. This limits the ability of the nodes to run efficient crawling strategies. To address this shortcoming, *GDist-RIA Crawler* [22] offers a coordinator-based approach to run the crawling strategy. It calculates the tasks to be done, and then dispatches the tasks to the nodes. Individual nodes execute the task and update the coordinator about their findings. Although this approach can apply any crawling strategy, it is not scalable, since the coordinator will eventually become a bottleneck as number of nodes increases.

In this paper, we propose *PDist-RIA Crawler*, a peer-to-peer and scalable architecture to crawl RIAs. Unlike Dist-RIA and GDist-RIA crawlers, all nodes are peer and homogeneous in the PDist-RIA Crawler. Nodes broadcast transitions information to the other nodes, therefore efficient crawling strategies can be implemented in this architecture. Termination is handled through a peer-to-peer ring-based protocol.

This paper contributes to the literature of web crawling by enhancing previously published works: Dist-RIA and GDist-RIA Crawlers. Additionally, performance of different operations during the crawling of RIAs are measured. These measurements are used to justify some of the decisions made in designing PDist-RIA Crawler. Finally, an implementation was used to get performance measurements.

We assume that nodes are independent from each other and there is no shared memory. Nodes can communicate between each other through message passing. Both the nodes and the communication medium are reliable. We assume that the target RIA is deterministic, that is: execution of an event from a state always leads to the same target state. We assume that the number of events and the number of states in the web application are finite. Finally, we assume that on average, there are more events per state than there are nodes in the system, i.e. the application graph is dense.

The rest of this paper is organized as follow: In Section 2 we describe some of the related works. In Section 3 we describe PDist-RIA Crawler. In Section 4 we describe some experimental results on the time it takes to perform different operations. In Section 5 we evaluate the performance of PDist-RIA Crawler against GDist-RIA Crawler. Finally, this paper is concluded in Section 6.

2 Related Works

Web crawling has a long and interesting history [23, 25]. Parallel crawling of the web is the topic of extensive research in the literature [3, 4, 12, 16, 18]. Parallel crawling of RIAs, on the other hand, is a new field. Dist-RIA crawler [24] studies running breath-first search in parallel. GDis-RIA crawler [22] uses a centralized architecture to run the greedy strategy and to dispatch jobs to other nodes. Hafaiedh et al. [15] propose a fault tolerant ring-based architecture to crawl RIAs concurrently using the greedy strategy with multiple coordinators. To the best of our knowledge, the PDist-RIA Crawler is the first work that does not have coordinators and where nodes are homogeneous.

Using efficient crawling strategies is another way to reduce the time it takes to crawl RIAs. Duda et al. studied the breath-first search strategy [11, 13, 19], and Mesbah et al. studied the depth-first search strategy [20, 21]. *Model Based Crawling* (MBC) increases the efficiency of the crawler by using a model of the application to choose the next task to execute. Examples of MBC are *Hypercube model* [2, 9], *Menu model* and *Probabilistic model* [6, 8, 10].

Identification of client-side states is an important aspect of crawling RIAs. Strict equivalence of DOMs is determined through hashing the serialized DOM [11, 13, 19]. Less strict approaches use an edit distance [20, 21] and elements on the page [1]. In this paper we use strict equivalence of DOMs and use hash of serialized DOM to identify the state.

Ranking states and pages is an approach to better utilize limited resources available in crawling a website [25]. In the context of traditional web applications, the *PageRank* [5] algorithm emulates a user session, calculates the probability

of a hypothetical user reaching a page, and uses this probability to rank a page. The *AjaxRank* [13] algorithm applies the same technique to RIA crawling. In this paper we assume that all states have the same rank.

3 Overview of the PDist-RIA Crawler

Nodes of the PDist-RIA Crawler partition the search space through events in the page (the same way as in Dist-RIA Crawler). Nodes divide the events in each state deterministically and autonomously, such that every event belongs to one node. Nodes are responsible to go to all states and execute their events. New states and transitions, discovered through executing events, are shared with all other nodes through broadcasting.

3.1 Algorithm

As depicted in Figure 1, nodes in the PDist-RIA Crawler can be in one of the following states: *Initial*, *Awake*, *Working*, *Idle*, and *Terminated*. Nodes start in the *Initial* state. In this state, nodes start up a headless browser process. The crawling starts when the node with node identifier 0 broadcasts a message that moves all nodes to the *Working* state. In this state nodes start running the crawling algorithm. Nodes find the next event to execute locally and deterministically.

If a node has nothing to do it goes to the *Idle* state. In this state the node waits for the termination state token or a new state. If a new state becomes available, it goes back to the *Working* state. If the termination token arrives, the node runs the termination algorithm to determine the termination status. The termination algorithm is described in detail in the next section.

3.2 Termination

A peer-to-peer protocol is run to determine termination. The protocol runs along with the crawling algorithm throughout the crawling phase. This protocol works by passing a token called *termination state* token around a ring overlay network that goes through all nodes. The termination token contains the following objects:

- List of state IDs for discovered application states: This list has an element per discovered state. As the token goes around the ring, more state IDs are added to this list.
- The number of known application states for each node: When the token visits a node, the node counts the number of application states it knows and stores this number in this list.

Using the information stored in the token, the termination algorithm described below decides whether to pass the token to the next node, or initiate termination.

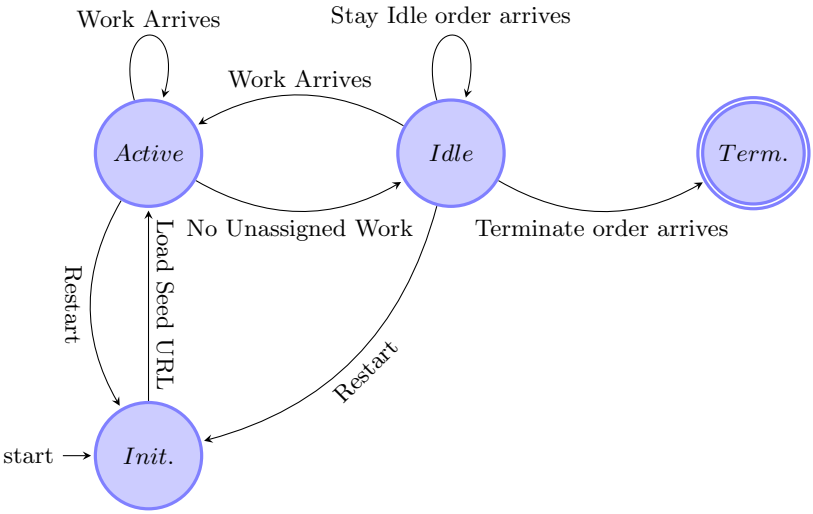

Fig. 1. The Node Status state diagram

Termination Algorithm. When the token arrives at a node, the node is either in Working state or in Idle state. If it is in Working state it will continue executing tasks and hold on to the token until it goes into the Idle state. In the Idle state the node performs the following steps:

1. The node updates the token with the new application states it knows about that do not yet exist in the token.
2. The node updates the number of the states it knows about.
3. If the status of the node is not indicated to be Idle in the token, the node updates its status to Idle in the token. This situation happens if this is the very first time the node takes the token. Initially, the status of all nodes is set to Active in the token. As the token goes around the ring, it can only pass a node if the node is in the idle state. Thus after one round of going around the ring, all node statuses will be Idle.
4. The node loops through the list of node states in the token and if it finds at least one node that is in Working state, the node passes the token to the next node in the ring.
5. If all nodes are in the Idle state in the token, the node loops through the list of number of known states for all nodes in the token and compares the number of known states for all nodes against the number of application state IDs in the token. If there is at least one node that does not know about all states discovered, the node passes the token to the next node.
6. If the last two steps do not pass the token to the next node, the node concludes that crawling is over and it initiates a termination by broadcasting a termination order to all the nodes.

Proof of Correctness: Let us assume that the algorithm is not correct and the termination is initiated while there are still events to be executed. Without

loss of generality, let us assume that node A initiated the termination order. The termination can only start if the token goes around the ring at least once and finds out that all nodes are idle and all nodes know about all states. For the termination to be wrong, let us assume that there is at least one event to be executed by a node, say node B. The termination order cannot be initiated if the token indicates that node B is not in the idle state. Thus, node B was in idle state when the token visited it after node B passed the token to the next node, a message was sent to it with a new state. Let us call the sender of the message node C. Node C can either be one of the nodes that the token visited on its way from node B to node A, or one of the nodes outside this path.

Node C cannot be one of the nodes that the termination token visited on its way from node B to node A. If that was the case, on its visit to node C the new state would be added to the list of application states in the token, the termination order would not be initiated by node A since at least the number of states known by node B is lower than the number of application states known in the token. So node C is not visited by the token on its way from node B to node A.

For the same reason that was stated for node B, node C was idle at the time when the token visited it and another node sent it a message with a new state. The sender, henceforth referred to as node D cannot be on the way from node C to node A, for the same reason that node C cannot be on the way from node B to node A.

This reasoning does not stop at node D and it continues indefinitely. Since the number of nodes that are not on the way of the token from the sender node to node A is finite eventually we run out of nodes to be potential senders, and thus the initial message telling node B about a new state could have never been initiated. Thus the termination algorithm is proven to be correct by contradiction.

4 Performance Measurements

In this section we measure the performance of certain operations used by the crawling algorithm. These measurements are used to justify some of the design decisions made in Section 3.2.

4.1 Time to Transmit Messages

In PDist-RIA Crawler, communication happens through message passing. In this section we measure the efficiency of message passing. Figure 2 shows stack-bars of the time it takes to send a message from one node to another as a function of message size, in logarithmic scale. Each message was sent 100 times and the distribution of the time is demonstrated by the corresponding stack-bar.

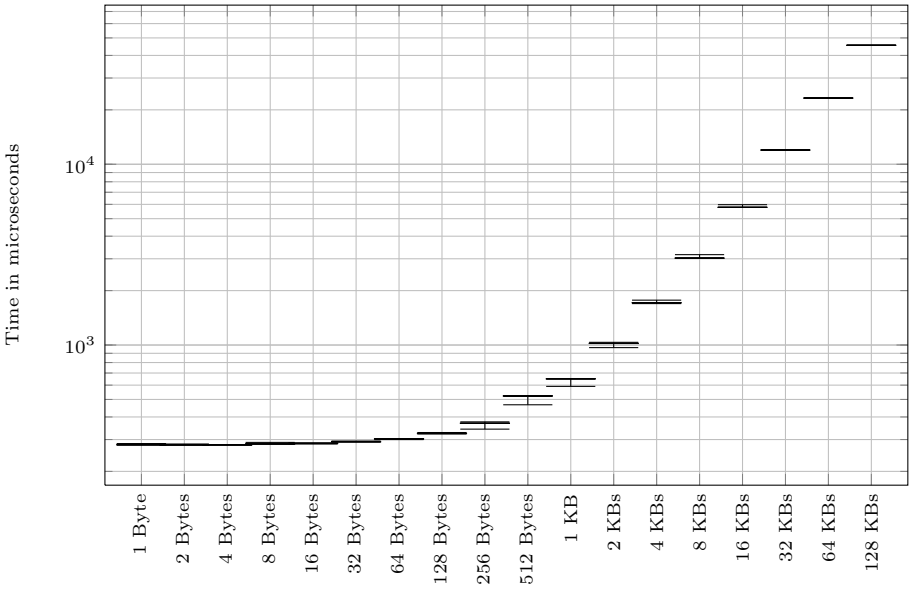

Fig. 2. Cost of sending messages between nodes

Given the measured network delays, we can calculate the overhead of sending different messages during the crawling process:

- **Overhead of sending state information:** A message to inform another node about a new state discovery contains state identifiers, number of events, a parent state identifier, and an event identifier in the parent state that leads to the discovered state. These items, along with the message header take between 128 to 256 bytes. Thus on average, the network delay to inform another node about a state is from 324 to 369 microseconds.
- **Overhead of sending transition information:** A message to inform another node about a transition contains source identifiers, target state identifiers, and event identifier. Similar to the *State* message, a transition message takes between 128 to 256 bytes. Thus on average network delay to inform another node about a transition is expected to be from 324 to 369 microseconds.
- **Termination Token:** The size of termination token varies depending on the number of crawler nodes, the number of discovered states, and the number of visited states by crawler nodes. In the worst case, the token contains state identifiers for all application states, and all nodes have visited all states. As explained in Section 3.2, among the test applications used in this paper, *Dyna-Table* with 448 states has the largest number of application states. Assuming we are crawling this application with 20 nodes, the termination token can get as large as 8 kilobytes. Thus in the worst case scenario where the token is at its largest size, the average time it takes to send the token from a node to another is less than 3 milliseconds.

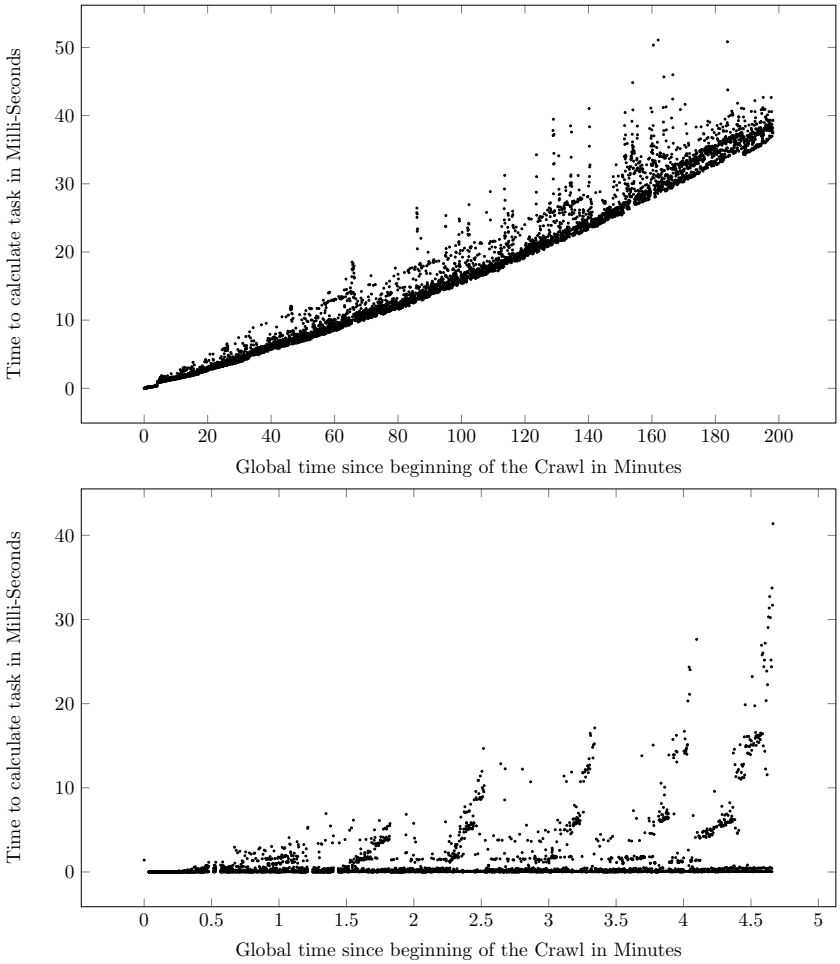

Fig. 3. Time to calculate next to execute as crawling proceeds using Breath-First (top) and Greedy (bottom) strategies for Dyna-Table application

4.2 Time to Calculate the Task to Execute

When the crawling strategy specifies a target event to execute, the crawlers often have to execute additional events to reach to the application state where the target event resides. We call this path of events and the target event a *task*. After execution of an event, the crawler has to calculate the next task to execute. Different crawling strategies run different algorithms to calculate the next task to execute. In this section we measure the time it takes to calculate the next task to execute for different crawling strategies.

Figure 3 shows the time it takes to calculate the next task using breath-first search and greedy strategies. In this figure, the y-axis shows the time to calculate

the next event, and the x-axis represents the clock since the beginning of the crawl. As the figure shows:

- The time to calculate the next task to execute, tends to rise steadily in the case of the breath-first search strategy. In this algorithm, after finishing each task, the crawler looks for the next task to execute by looking into the seed URL and then the most immediate children for a task to perform. As the crawl proceeds the algorithm should go deeper in the graph before it can find a new task. Thus the cost of running the algorithm tends to increase as the crawl proceeds.
- The time it takes to calculate the next task to execute is generally lower in the greedy algorithm. In this algorithm, the crawler does not have to start from the seed URL and check all immediate children before going to further children for a task. Thus the greedy algorithm has a higher chance of finding tasks earlier.

As Figure 3 shows, in the majority of the cases it takes a few milliseconds to calculate the next task to execute.

4.3 Number of Events in Tasks

Executing the task which often involves executing several client-side events, possibly interacting with the server, and possibly performing a reset, often takes much longer than calculating the next task to execute. In this section we measure the number of events in the tasks calculated by different crawling strategy.

Different crawling strategies create tasks with different number of events. Execution of an event can start an interaction with the application server. This interaction (often in form of an asynchronous request to the server) is often the most time consuming aspect of executing the task. Therefore, the number of events in each task is a good indicator of the time it takes to execute the task. By measuring the number of events in the tasks, in effect we forecast the time it takes to execute the tasks.

Figure 4 shows the number of events in tasks created using breath-first search and greedy strategies. In these figures, the y-axis shows the number of events, and the x-axis represents the clock since the beginning of the crawl. As the figures show:

- As the crawling proceeds the length of events in tasks increases using the breath-first search strategy. In Dyna-Table application this number can be as high as 14 events.
- The greedy strategy represent a very efficient strategy, where often very few events exist in each task. The rare worst case scenario happens with 7 events in a task.

The time it takes to execute individual events depends highly on the target application and the server hosting it. Execution of JavaScript events, that do not trigger an asynchronous call to the server, is substantially faster than the events that interact with the server. For example, in the Dyna-Table web application,

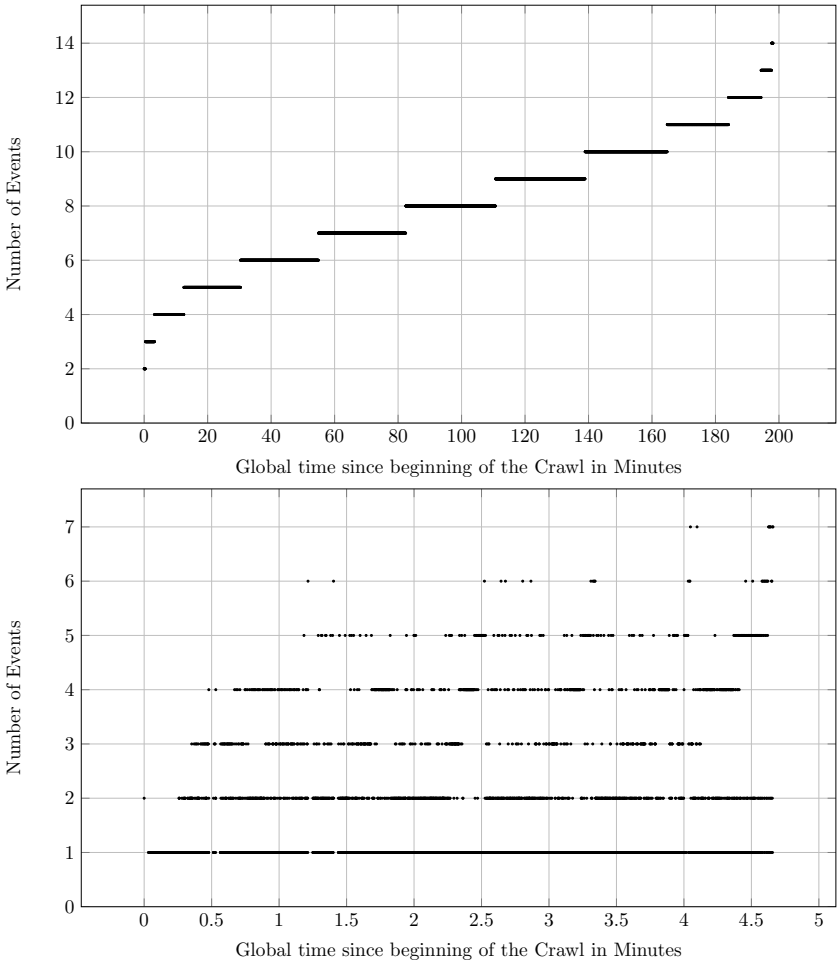

Fig. 4. Number of events to execute before executing a new event as crawling proceeds using Breath-First (top) and Greedy (bottom) strategies, for Dyna-Table application

execution of events that do not interact with the server always take less than 20 milliseconds. In the same application, events that do interact with the server often take more than 85 milliseconds to execute, and can take up to 1.3 seconds.

As the experimental results in the section show:

- The time required for communicating a state or transition between two nodes is less than a millisecond.
- The time required for calculating the next event to execute is often less than 100 milliseconds.
- The time required for executing a JavaScript event is often between 10 to 1000 milliseconds. A task is often composed of several JavaScript events, as the result executing a task can take up to several seconds.

The time it takes to broadcast a message is orders of magnitude smaller than the time it takes to calculate a task or execute it. It is thus reasonable to expect a good performance from a peer-to-peer architecture where nodes broadcast states and, when necessary, transitions. As the number of nodes increases, the number of messages per second increases too, and the network is bound to become a bottleneck. For example, based on the experimental measurements presented, we expect that when both states and transitions are broadcasted, by crawling the Dyna-Table application with 434 nodes or more, the network becomes a bottleneck. Before reaching this point, however, a good performance speedup is expected with this architecture.

Based on this observation, we devise a peer-to-peer architecture. This architecture takes advantage of low network delay and relies on broadcasting the information needed. Assuming that the number of events per state is larger than the number of nodes, through elimination of any centralized unit, this architecture does achieve a better scalability.

5 Evaluation

5.1 Test-Bed

For the experimental results discussed in this chapter, the nodes and the coordinator are implemented as follow: The JavaScript engine in the nodes is implemented using PhantomJS 1.9.2. Strategies are implemented in the C programming language and GCC version 4.4.7 is used to compile them. All crawlers use the Message Passing Interface (MPI) [27] as the communication mechanism. MPI is an open standard communication middleware developed by a group of researchers with background both in Academia and industry. MPI aims at creating a communication system that is interoperable and portable across a wide variety of hardware and software platforms. Efficient, scalable, and open source implementations of MPI are available such as OpenMPI [14] and MPICH [17]. MPICH version 3.0.4 is used to implement the communication channel in our experiments. All nodes, as well as the coordinator in case of GDist-RIA crawlers, run on Linux kernel 2.6.

The nodes are hosted on Intel® Core(™)2 Duo CPU E8400 @ 3.00GHz and 3GB of RAM. The coordinator is hosted on Intel® Xeon® CPU X5675 @ 3.07GHz and 24GB of RAM. The communication happens over a 10 Gbps local area network. We ran each experiment three times and the presented numbers are the average of those runs.

To compare the relative performance of the crawlers, we implemented GDist-RIA and PDist-RIA crawlers using the same programming language and used MPI as communication channel. In all cases the C programming language is used to calculate the next task to do. In order to compare the performance of crawlers we crawled two different target web applications using the two architectures. In both cases we used the breath-first search and the greedy strategies.

5.2 Target Applications

Two real world target applications (Figure 5) are chosen to measure the performance of the crawlers. The target web applications are chosen based on their size, complexity and client side features they use.

Dyna-Table is a real world example of a JavaScript widget, with asynchronous call ability, that is incorporated into larger RIAs. This widget helps developers to handle large interactive tables. It allows to show a fixed number of rows per page, to navigate through different pages, to filter content of a table based on given criteria, and to sort the rows based on different fields. This application was developed using the *Google Web Toolkit* and has 448 states and a total of $5,380$ events.

Periodic-Table is an educational open source application that simulates an interactive periodic table. This application allows the user to click on each element and show the user information about the element in a pop-up window. The application can display the periodic table in two modes: the small mode, and the large mode. The two modes are identical in terms of functionality, however, they offer two very different interfaces. Once an element is clicked and the pop-up window shows up for the element, other elements can be clicked or the pop-up window can be closed. This application is developed in PHP and JavaScript, and it has 240 states and $29,040$ events.

Table 1 shows some information about the graph of the target web applications. As the table shows: Dyna-Table is a large size RIA with a small number of events per page, and Periodic-Table is a large size RIA with a large number of events per page. In Periodic-Table, all states have more events per page than the number of web crawlers and the application graph is dense.

Table 1. Target Applications graph summary

Application Name	Number of States	Number of Transitions	Average Events per Page
Dyna-Table	448	5,380	12.01
Periodic-Table	240	29,040	121.00

5.3 Results

Figure 6 shows the time it takes to crawl the Dyna-Table and Periodic-Table applications with different number of nodes using different crawlers. As the figure shows the overhead of the peer-to-peer architecture is not tangible. In fact, even with 20 nodes, the GDist-RIA Crawler does not scale as well as PDist-RIA Crawler. This is particularly noticeable for the greedy strategy. In this strategy, tasks take less time to execute, and therefore nodes ask the coordinator for new tasks to execute more frequently, making the coordinator a bottleneck. Therefore, the new method is not less efficient than the best known method to date, when that best known method to date is not overloaded. In additionally, the new method beats the old method squarely when the other becomes overloaded.

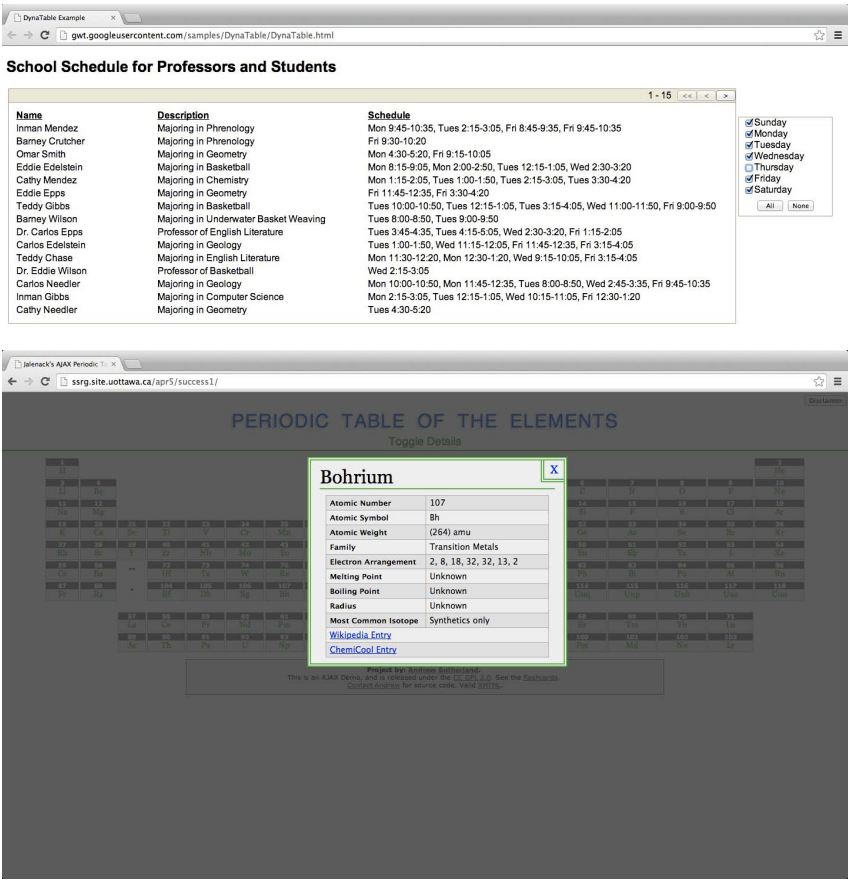

Fig. 5. Target Application: Dyna-Table (up) and Periodic-Table (down)

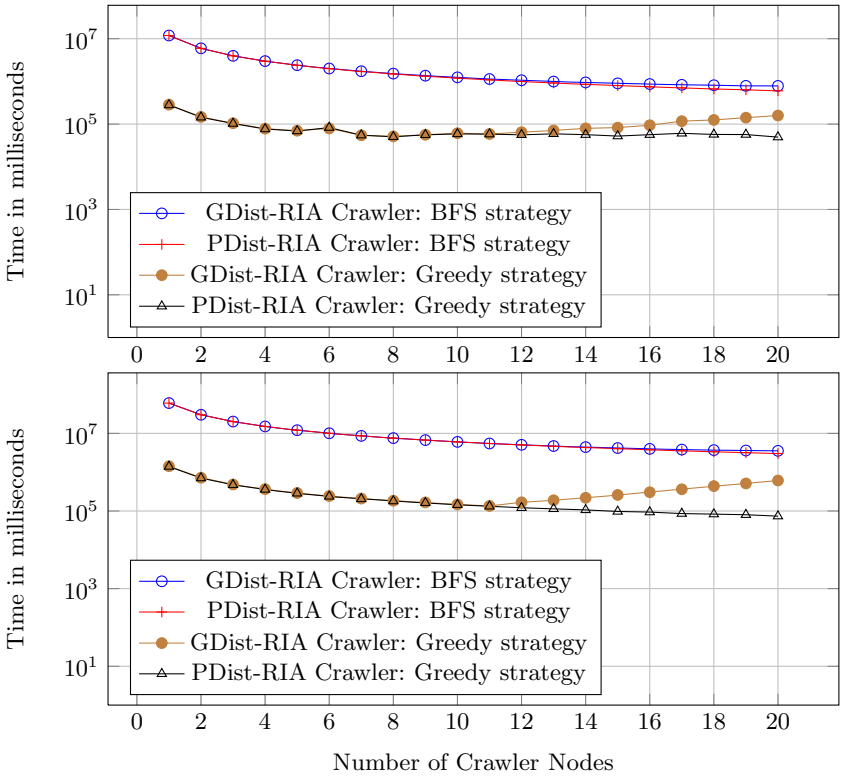

Fig. 6. The total time to Crawl Dyna-Table (top) and Periodic-Table (bottom) in parallel using using different architectures

6 Conclusion and Future Improvements

In this paper we introduced PDist-RIA Crawler, a peer-to-peer crawler to crawl RIAs. To design this crawler, we measured the time it takes to execute different operations, and based on the measured values we engineered the PDist-RIA Crawler. This crawler was implemented and its performance was compared against the GDist-RIA crawler.

In this paper, we assumed that new transitions are broadcasted as soon as they become available. A study of the impact of this assumption is missing from this paper. More formally, the impact of the following two assumptions is missing: Firstly, to utilize the network better, it may be more efficient not to broadcast the new transitions as they become available, but to broadcast them in batches, or broadcast them at given time intervals. Secondly, not all transitions have a major impact on reducing the time it takes to crawl the RIA. Sharing only a sub-set of transitions, instead of all transitions, may not increase the time it takes to crawl the RIA, while it reduces network traffic.

Acknowledgements. This work is largely supported by the IBM® Center for Advanced Studies, the IBM Ottawa Lab and the Natural Sciences and Engineering Research Council of Canada (NSERC). A special thank to Sara Baghbanzadeh.

References

1. Amalfitano, D., Fasolino, A.R., Tramontana, P.: Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications. In: IEEE International Conference on Software Maintenance, ICSM 2009, pp. 571–574 (September 2009)
2. Benjamin, K., von Bochmann, G., Dincturk, M.E., Jourdan, G.-V., Onut, I.V.: A strategy for efficient crawling of rich internet applications. In: Auer, S., Díaz, O., Papadopoulos, G.A. (eds.) ICWE 2011. LNCS, vol. 6757, pp. 74–89. Springer, Heidelberg (2011)
3. Boldi, P., Codenotti, B., Santini, M., Vigna, S.: Ubicrawler: A scalable fully distributed web crawler. In: Proc. Australian World Wide Web Conference, vol. 34(8), pp. 711–726 (2002)
4. Boldi, P., Marino, A., Santini, M., Vigna, S.: Bubing: Massive crawling for the masses
5. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the Seventh International Conference on World Wide Web 7, WWW7, pp. 107–117. Elsevier Science Publishers B. V, Amsterdam (1998)
6. Choudhary, S., Dincturk, E., Mirtaheri, S., Bochmann, G.V., Jourdan, G.-V., Onut, V.: Model-based rich internet applications crawling: Menu and probability models
7. Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Jourdan, G.-V., Bochmann, G.v., Onut, I.V.: Building rich internet applications models: Example of a better strategy. In: Daniel, F., Dolog, P., Li, Q. (eds.) ICWE 2013. LNCS, vol. 7977, pp. 291–305. Springer, Heidelberg (2013)
8. Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Jourdan, G.-V., Bochmann, G.v., Onut, I.V.: Building rich internet applications models: Example of a better strategy. In: Daniel, F., Dolog, P., Li, Q. (eds.) ICWE 2013. LNCS, vol. 7977, pp. 291–305. Springer, Heidelberg (2013)
9. Dincturk, E., Jourdan, G.-V., Bochmann, G.V., Onut, V.: A model-based approach for crawling rich internet applications. ACM Transactions on the Web (2014)
10. Dincturk, M.E., Choudhary, S., von Bochmann, G., Jourdan, G.-V., Onut, I.V.: A statistical approach for efficient crawling of rich internet applications. In: Brambilla, M., Tokuda, T., Tolksdorf, R. (eds.) ICWE 2012. LNCS, vol. 7387, pp. 362–369. Springer, Heidelberg (2012)
11. Duda, C., Frey, G., Kossmann, D., Matter, R., Zhou, C.: Ajax crawl: Making ajax applications searchable. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE 2009, pp. 78–89. IEEE Computer Society, Washington, DC (2009)
12. Edwards, J., McCurley, K., Tomlin, J.: An adaptive model for optimizing performance of an incremental web crawler (2001)
13. Frey, G.: Indexing ajax web applications. Master's thesis, ETH Zurich (2007), `http://e-collection.library.ethz.ch/eserv/eth:30111/eth-30111-01.pdf`
14. Gabriel, E., et al.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)

15. Hafaiedh, K., Bochmann, G., Jourdan, G.-V., Onut, I.: A scalable p2p ria crawling system with partial knowledge (2014)
16. Heydon, A., Najork, M.: Mercator: A scalable, extensible web crawler. World Wide Web 2, 219–229 (1999)
17. Karonis, N.T., Toonen, B., Foster, I.: Mpich-g2: A grid-enabled implementation of the message passing interface. Journal of Parallel and Distributed Computing 63(5), 551–563 (2003)
18. Li, J., Loo, B., Hellerstein, J., Kaashoek, M., Karger, D., Morris, R.: On the feasibility of peer-to-peer web indexing and search. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 207–215. Springer, Heidelberg (2003)
19. Matter, R.: Ajax crawl: Making ajax applications searchable. Master's thesis, ETH Zurich (2008), http://e-collection.library.ethz.ch/eserv/eth:30709/eth-30709-01.pdf
20. Mesbah, A., Bozdag, E., Deursen, A.V.: Crawling ajax by inferring user interface state changes. In: Proceedings of the 2008 Eighth International Conference on Web Engineering, ICWE 2008, pp. 122–134. IEEE Computer Society Press, Washington, DC (2008)
21. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based web applications through dynamic analysis of user interface state changes. TWEB 6(1), 3 (2012)
22. Mirtaheri, S.M., Bochmann, G.V., Jourdan, G.-V., Onut, I.V.: Gdist-ria crawler: A greedy distributed crawler for rich internet applications
23. Mirtaheri, S.M., Dinçtürk, M.E., Hooshmand, S., Bochmann, G.V., Jourdan, G.-V., Onut, I.V.: A brief history of web crawlers. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, pp. 40–54. IBM Corp. (2013)
24. Mirtaheri, S.M., Zou, D., Bochmann, G.V., Jourdan, G.-V.,, I.V.: Dist-ria crawler: A distributed crawler for rich internet applications. In: 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pp. 105–112. IEEE (2013)
25. Olston, C., Najork, M.: Web crawling. Foundations and Trends in Information Retrieval 4(3), 175–246 (2010)
26. Peng, Z., He, N., Jiang, C., Li, Z., Xu, L., Li, Y., Ren, Y.: Graph-based ajax crawl: Mining data from rich internet applications. In: 2012 International Conference on Computer Science and Electronics Engineering (ICCSEE), vol. 3, pp. 590–594 (March 2012)
27. Snir, M., Otto, S.W., Walker, D.W., Dongarra, J., Huss-Lederman, S.: MPI: the complete reference. MIT Press (1995)